Engineering with Computers
DOI 10.1007/s00366-007-0071-0

ORIGINAL ARTICLE

Measuring progress in Premo order-verification

Ryan B. Bond · Curtis C. Ober · Patrick M. Knupp

Received: 15 November 2006 / Accepted: 20 January 2007

Abstract Since verification of computational simulation codes requires significant resources, the ability to measure progress in verification is critical to assess whether resources are being applied appropriately. Additionally, potential users need to know what fraction of the software has been order-verified. In this study, the procedures and progress measures presented by Knupp et al. (Measuring progress order-verification within software development projects. Engineering with Computers, appears in this issue, 2007) are demonstrated on the Premo software, which simulates compressible aerodynamics through and around general geometries. Premo was selected for this demonstration because extensive order-verification tests have been previously performed, yet no systematic effort has been made to assess test-suite completeness or progress. This effort was performed to identify the practical issues encountered when attempting to apply the ideas by Knupp (Measuring progress order-verification within software development projects. Engineering with Computers, appears in this issue, 2007) to existing production-quality software. In this work, a non-specific order-verification exercise is considered, as opposed to an application-specific order-verification exercise, since past and present Premo order-verification efforts have been motivated by the need to verify all of the code, rather than portions relevant for specific applications. Constructing an order-verification test suite that verifies the order of accuracy of all the code capabilities is a major step in measuring progress. A practical approach to test-suite construction is described that helps create a complete test suite through a combination of coarse-grain code coverage, input-keyword inspection, discretization-algorithm documentation, and expert knowledge. Some of the difficulties and issues encountered during the construction of the test suite are described, along with recommendations on how to deal with them. Once the test suite is constructed, the progress measures proposed by Knupp (Measuring progress order-verification within software development projects. Engineering with Computers, appears in this issue, 2007) can be evaluated and used to reconstruct the history of progress in Premo verification over the past several years. Gaps in Premo verification are identified and indicate future directions for making progress. Additionally, a measure of Premo verification fitness is computed for selected applications commonly simulated in the aerospace industry. It is hoped that this demonstration will provide a practical example for other software-development groups in measuring their own verification progress.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

R. B. Bond (✉)
Aerosciences, Dept. 1515, Sandia National Laboratories,
P.O. Box 5800, MS 0825, Albuquerque, NM 87185, USA
e-mail: rbbond@sandia.gov

C. C. Ober
Exploratory Simulation Technologies, Dept. 1433,
Sandia National Laboratories, P.O. Box 5800,
MS 0316, Albuquerque, NM 87185, USA
e-mail: ccober@sandia.gov

P. M. Knupp
Optimization & Uncertainty Estimation, Dept. 1411,
Sandia National Laboratories, P.O. Box 5800,
MS 0316, Albuquerque, NM 87185, USA
e-mail: pknupp@sandia.gov

1 Introduction

Verifying the order of accuracy (OA) of computational codes is an extremely resource intensive task. The development and testing of progress measures are essential steps in determining whether these resources are being applied appropriately, and progress measures can bee used to indicate to potential users what fraction of the software that has been order-verified. The theoretical foundations for how to construct a complete test suite for order verification, how to measure progress in demonstrating the order behavior of a code using this test suite, and how to relate this information to specific applications of a code are presented in [1]. It is the first known attempt at documenting a process for completing order-verification tasks for a production-quality code. The current work is intended to be a practical application of some of the concepts presented in [1], although not all of the concepts from [1] are included in this exercise. The computational-fluid-dynamics code Premo [2] and its past and present order-verification attempts are used to document a practical example of the task of generating an order-verification test suite, measuring progress in code order verification, and measuring the readiness of a code (relative to its order-verification status) for a particular application. The challenges encountered when applying the ideas from [1] are documented. When general concepts from [1] must be applied for the specific case of Premo, the details have been included, and when certain assumptions or simplifications must be made to reduce the scope of this initial example exercise, those assumptions and simplifications have also been stated.

Premo is a compressible fluid dynamics code being developed at Sandia National Laboratories. It is used to determine aerodynamic performance of complex geometries. It is a parallel, unstructured, edge-based, finite-volume code. Details about Premo are presented only when necessary—a complete discussion of Premo can be found in [2].

Numerous terms have been introduced in [1] and will not be re-defined here since [1] appears in this same issue. However the definitions of a couple of additional key terms are given here to clarify distinctions between different algorithms.

ordered algorithm: any algorithm with an associated OA.

order-affecting algorithm: any calculation which, if performed incorrectly, would adversely affect the observed OA. All ordered algorithms are, by definition, order-affecting algorithms. The set of ordered algorithms is a subset of the set of order-affecting algorithms.

Examples of ordered and order-affecting algorithms are as follows:

1. The numerical differentiation of some variable in a finite-difference calculation is an ordered algorithm (and thus also an order-affecting algorithm).
2. The calculation of viscosity, μ, via some algebraic law in a Navier-Stokes code is not an ordered algorithm, but it is an order-affecting algorithm. Although the formula for μ may be algebraic, the result of its calculation is embedded within a numerical differential operation with an associated OA. Therefore, the observed OA of the numerical differentiation would be affected by an incorrect μ calculation.

The Premo [2] software was chosen for this work because of the maturity of its verification efforts in both temporal and spatial discretizations. Order verification (OV) for the temporal-discretization terms in Premo has included verification by the method of exact solutions using a convecting-vortex problem. Recent OV efforts for spatial-discretization terms (both interior equation sets and boundary conditions (BCs)) have been performed using the method of manufactured solutions (MMS) [3–5]. Earlier work is described in [6, 7]. To date, several coding mistakes and algorithmic weaknesses have been found and corrected through code OV.

In this work, the focus is restricted to the initial attempt at verifying the order of accuracy of code features, rather than ongoing, regression-style testing meant to preserve order behavior during continuous code development. This is a closed-ended process, presumably taking place once for each version release. Testing of individual functions (i.e., unit testing) upon initial implementation of features is not addressed in this paper because it is not part of OV as presented in [1]. These issues are extremely important, but fall outside of the scope of the present study. A key purpose of this study is to create a database of order-of-accuracy tests (OATs) and results such that different measures (those conceived in [1] as well as others) can be evaluated over time. Such a database allows progress-measure definitions to evolve so that they are representative both of effort level on OA verification and of the readiness of a code to be used for a particular application. This study is an example of how to implement progress-measure theory; different codes may need to follow slightly different procedures, even though the concepts should generally apply to a wide spectrum of OA codes.

The set of all code lines (a portion of software) that affect the OA is referred to as the ‘‘OV domain’’, and is given the symbol Ω. It can be decomposed into a developer-oriented partition, such as functions or lines, or a user-oriented decomposition, such as code options or features. (Here we can not use the term ‘‘partition’’ for the case of options, since different code options may have overlapping lines of code.) A particular domain, $\Pi \subseteq \Omega$, is the subset of code lines that affect the OA relative to a particular order-verification exercise (OVE), OAT, application, etc. Two key concepts presented in [1] are the ''progress measure'' and ''fitness measure''. A progress measure is a metric which communicates the progress in an OVE over time. For the purpose of this study, only a non-specific OVE is considered, where the OVTS is complete relative to Ω, rather than the case of an application-specific OVE, where the OVTS is designed to cover a subset of the OV domain associated with a specific application, $\Pi_{\text{app.}} \subset \Omega$. If the OV status of a code relative to a specific application is needed, but no application-specific OVE has been performed, then a fitness measure indicates the OV status relative to the application using data from a non-specific OVE. Numerous progress and fitness measures are proposed in [1], and some of them are calculated for Premo in this work. Section 3 gives the definitions and abbreviated discussions of these measures.

A principal challenge of measuring OV progress is the design of the OVTS. If the *exact* mappings between the governing equations and the corresponding portions of the software (functions, lines of code, etc.) are documented, then construction of the OVTS is significantly simplified. However, such documentation is often non-existent or incomplete. A complete OVTS is one that tests, in some meaningful sense, the *full* range of an OA code's capabilities. The completeness of an OVTS can be relative to a given version of a code and, if necessary, relative to a set or subset of features of that version of the code. Thus, each OVE has an OVTS which is complete relative to a particular domain, $\Pi_{\text{OVE}} \subseteq \Omega$. For example, an OVTS can be designed to thoroughly test all documented or mature features ($\Pi_{\text{doc.\&mat.}}$) in a given version but to not test some experimental or undocumented features which are still being developed in anticipation of future releases. This is what was done in the present work; thus, $\Pi_{\text{OVE}} = \Pi_{\text{doc.\&mat.}} \subset \Omega$. Once the particular domain for the OVE, Π_{OVE}, has been established, the OVTS must be evaluated for completeness so that gaps in OVTS coverage can be identified and addressed. Additionally, a quantitative indicator of OVTS completeness can be evaluated over time, so that the process of OVTS design and construction can be monitored (although we do not attempt to do so in the present work). We propose several different techniques for evaluating OVTS completeness in Sect. 2.2.

2 Construction of the OVTS

As described in [1], the process of constructing an OVTS to cover Π_{OVE} is iterative:

1. Start with an initial OAT, ϕ_1, that covers some fraction of the particular domain for the OVE, Π_{OVE};
2. For $n \geq 1$, evaluate the OVTS, $\{\phi_1,\ldots,\phi_n\}$, to determine what fraction of Π_{OVE} is not covered;
3. Introduce a new OAT, ϕ_{n+1}, which covers some portion of Π_{OVE} missed by $\{\phi_1,\ldots,\phi_n\}$;
4. Continue until no candidate OATs can be conceived which cover anything new.

This process does not result in a unique OVTS. The final OVTS is dependent upon the initial OAT, the way(s) in which coverage completeness is evaluated, the sequence in which new OATs are introduced, and the desired granularity. Since measures for verification progress are based upon the results of each OAT in the OVTS, any given progress measure will carry some dependence upon the OVTS and thus inherit its non-uniqueness. This is permissible since the intent of progress measures is to quantify progress in verification over time, so any progress measure will increase over time as long as the code and the OVTS remain static. The dynamic case, where the code is still in development and the OVTS is evolving with it, while important for practical application, is beyond the scope of the present discussion.[1] The following sections give in-depth explanations of the steps in OVTS construction.

2.1 Initial OVTS

The initial OVTS can be a single OAT created from scratch or collection of OATs from prior OV efforts. In the case of this study, the initial OVTS consisted of seven inviscid convecting-vortex tests (an unsteady exact solution problem), each using different options for the temporal integration, and nine steady MMS tests from [3–5] for spatial order verification of the interior equation sets and BCs. Some OATs from prior OV efforts (namely, some of [3–5] and all of [6, 7]) were omitted from the initial OVTS. This was done because they were redundant, relative to OATs that were part of the initial OVTS. An example of such redundancy are the 2D tests in [6, 7] which were superseded by 3D tests in [3–5]. The omission of these tests coarsened the granularity of the OVTS. A coarse-grained OVTS takes less time to run since it has fewer tests. However, a fine-grained OVTS more accurately shows incremental progress and is superior for isolating problems in the code. For example, the passing of the MMS tests from [6, 7] does not show up as an increase in our progress measure since our OVTS omits them. In practice, an OVTS will have a granularity somewhere between two extremes:

[1] This case is discussed in [1], but a thorough example study would require a series of code releases and associated OVTS versions, requiring several years and implementation of a more formal OV process for Premo.

the finest OVTS, in which each OAT tests only one unique code capability, and the coarsest OVTS which represents the minimum number of OATs meeting the coverage completeness requirement. An OVTS used for initial OV is likely to be more fine-grained. This allows for easy isolation of problems to certain code features or algorithms. An OVTS used for regression-style testing, after initial OV has been completed, is likely to be more coarse-grained, so that fewer tests need to be maintained and run.

2.2 Evaluating completeness of the OVTS

An OVTS is complete relative to Π_{OVE} if it tests all of the lines in Π_{OVE} with all the required generality. The OV domain can be broken down into many different decompositions. An end-user-oriented decomposition is one that defines the OV domain in terms of code options or features. A complete OVTS tests every possible option or feature contained in the particular domain associated with its OVE. A developer-oriented decomposition is one that defines the OV domain in terms of portions of the software: lines, blocks of lines, functions, groups of functions, etc. Developer-oriented decompositions usually group lines of software into subsets; such decompositions are usually partitions. End-user-oriented decompositions are frequently not partitions, since the subsets of code lines associated with different inputs or features are not necessarily disjoint. A complete OVTS tests each piece of the software contained in the particular domain associated with its OVE in its most general fashion. As mentioned previously, the OV domain of interest can be restricted to features (or portions of software) that are considered to be mature, documented, and advertised for general use. In this study, we limited the investigation to the mature and documented capabilities of version 1.3β of Premo, so that $\Pi_{\mathrm{OVE}} = \Pi_{\mathrm{doc.\&mat.,1.3}\beta} \subset \Omega$. Additionally, since Premo relies on several supporting pieces of software (the Sierra[2] framework [8] as well as third-party libraries (TPLs)), Π_{OVE} was taken to be that defined only by Premo itself, not all of the supporting software. Any end-to-end test of Premo obviously tests the supporting software for a specific case, but no effort was made to have the OVTS completely cover the OV domains mapped out by the Sierra framework or TPLs. This omission was done simply to reduce the scope of the present study. Obviously, to verify all of Premo, the OVTS must cover the features of Sierra and the TPLs that Premo uses, in the most general way that Premo can use them.

To evaluate the completeness of the OVTS, coverage indicators are needed. A coverage indicator is a function valued in the range [0,1] and can be associated with some decomposition of the OV domain. A value of 1 indicates that the set has been completely covered. Any given decomposition of the OV domain will have at least one associated coverage indicator. An example coverage indicator is line coverage, where the number of lines covered by the OVTS can be measured with some software development tools. Line coverage is not a particularly effective coverage indicator because it is impractical to sort out which lines belong to the OV domain and which do not.[3] Three primary coverage indicators are used for this study: expert knowledge, coarse-grained code coverage (at the function level), and input-keyword evaluation.

We first attempted using expert knowledge to document all the equations and capabilities in order to identify what should be in the OVTS. This became impractical after a while due to the huge number of capabilities and combinations. A key finding was that it was difficult to construct the complete test suite in this fashion because the code-documentation requirements are significant and most code documentation would be insufficient for this purpose. We recommend in the future that new codes provide documentation detailed enough to identify all code capabilities and options and their mappings to portions of the software, input options, and the governing equations. This is no small task and should be included as part of the development process.

Using expert knowledge to evaluate OVTS completeness is good for initial OVTS design, when broad categories of OATs are conceived and implemented, and can be based on the documentation of the code with respect to order-affecting algorithms. Once the OVTS has matured to a certain level, coarse-grained code coverage and input-keyword evaluation, both automated processes, can be used to detect additional holes in the code coverage. Expert knowledge misses many holes because it is an inherently subjective and labor-intensive process, which is prone to mistakes. Additionally, expert knowledge provides only a conceptual decomposition of the OV domain, and accordingly, can not be used to produce a quantitative indicator of OVTS completeness. However, expert knowledge is critical, since it must be used to ensure that all exercised code options and portions of software are tested with the necessary generality—something automated tools can not do without being extremely sophisticated and complex.

One of the automated methods for evaluating coverage of the OVTS is to trace the function coverage of code execution for each test and then compile that coverage for the full test suite (i.e., coarse-grained code coverage). One

[2] The Sierra framework software and its input options are independent of the governing equations, but they do handle such capabilities as coupling, adaptivity, and shape functions.

[3] For example, a line which prints a message to the user might be outside of the OV domain, whereas a line which calculates a numerical derivative would be inside the OV domain.

major advantage of this method is that it actually traces the code execution, allowing an audit of the perceived mapping of order-affecting algorithms to portions of software. Sometimes coding logic mistakes cause improper paths through the software, so that properly implemented lines are never executed. Such coding mistakes can actually be identified in the OVTS construction phase with the coarse-grained code-coverage indicator, even before the first OATs are executed on the full sequence of meshes. One disadvantage of any code-coverage-based indicator is that one can not know, without also incorporating expert knowledge, whether executed portions of the software have actually been tested with the required generality. Any code-coverage-based indicator must be implemented at a certain granularity. An advantage of a coarse granularity is that it is simpler to map the OV domain to larger portions of software (such as functions) than it is to smaller portions (such as lines).[4] However a weakness of using coarse-grained code coverage is that finer branches may be missed. For example, if the coverage is evaluated at the function level, there may be branches internal to the function which may not be covered by the OVTS, even though the function is covered. An example in this study involves the no-slip boundary condition, which has both isothermal (constant temperature) and adiabatic (zero heat flux) options. The isothermal/adiabatic branch is inside the function for no-slip BC enforcement, so an OAT which tests either of these options will register as having covered the function. Thus, the function-coverage indicator can not identify the hole in the OV domain left by an OVTS which tests only one of the no-slip options.[5]

Another automated method for evaluating coverage of the OVTS is input-keyword coverage. This method uses a script to parse the code input file associated with an OAT, looking for keywords associated with certain input parameters. The options being exercised for a given OAT are compared to the full list of code options to determine the level of coverage, and then these statistics are compiled for the complete OVTS. One advantage of this method is that it applies a user-oriented decomposition of the OV domain, since users conceptualize the verification process in terms of what options or features have been verified. One key disadvantage of the keyword-coverage indicator is that it does not distinguish between options which are tested sufficiently generally and options that are not. Another is that it does not actually trace the code execution, so there is no way of knowing that the portions of code associated with a given option are actually being executed. Since the keyword coverage involves a script used to parse the input files and compare the keywords to the complete list, the level of sophistication in this script determines its usefulness. A high degree of effort can be put into making sure that the script fully identifies the complete range of possible input keywords (which may interact, be hierarchical, or have default values when omitted) and map these possibilities to the OV domain. Alternatively, the input file syntax can be designed so that even a simple script can accomplish this task. In the case of Premo, there is an additional complication because some ‘‘input’’ is defined by the mesh or restart file, rather than by the keywords in the input file. The input file only contains the name of the mesh or restart file, so parsing it with a script will not reveal anything about the mesh topology.

Each of these coverage indicators has pros and cons, as summarized in Table 1 along with the line-coverage indicator. The important thing to note is that each indicator has its own weaknesses, and that certain types of holes will be missed by each of them. In essence, the weaknesses can be placed into two broad categories: coverage holes left unidentified because the underlying decomposition does not actually span the OV domain and coverage holes missed because the process of implementing the coverage indicator is subject to mistakes. Using multiple coverage indicators is viewed by the authors as the best way of detecting OVTS coverage holes with the fewest mistakes and least effort, and expert knowledge, function coverage, and keyword coverage are believed to provide the most useful information at a reasonable cost. As noted in Table 1, line coverage has similar pros and cons as function and keyword coverage but requires line-by-line mapping to the OV domain, which is labor intensive and therefore not included in this study.

2.3 Filling coverage holes in the OVTS

As coverage holes are identified in the OVTS, new OATs must be introduced to fill them and thus complete the OVTS. In general, two types of solutions are used for OATs: exact solutions and manufactured solutions. Since they do not require calculation and implementation of a source term, exact solutions are used where they exist and where they are sufficiently general to test all terms in the governing equations and BCs. Where no sufficiently general exact solutions exist, or in cases where coarse

[4] For this study, only the ‘‘apply’’ and ‘‘execute’’ functions for each object were actually tracked for coverage because these functions were considered to best map out the OV domain in the software. This eliminated constructors, destructors, error handling functions, etc. This distinction is somewhat subjective and is only relevant for Premo’s specific software organization.

[5] In this particular case, however, both expert-knowledge and input-keyword coverage indicators can be used to make a distinction between the adiabatic and isothermal branches.

Table 1 Summary of pros and cons for OVTS coverage indicators

Indicators	Pros	Cons
Expert knowledge	Best for initial design	Not automated
	Evaluates generality	Subjective
Function coverage	Traces code execution	Does not evaluate generality
	Automated	Misses finer-grained branches
Keyword coverage	User-oriented indicator	Does not evaluate generality
	Automated	Does not trace code execution
Line coverage	Finds *all* untested lines	Finds *all* untested lines
	Traces code execution	Does not evaluate generality
	Automated	Requires line-by-line mapping of OV domain

granularity is desired and one manufactured solution can take the place of multiple exact solutions, OATs involving manufactured solutions are used to fill the OVTS. Although progress in OVTS construction can be measured quantitatively as described in [1], no attempt was made to do this for Premo.

The full OV domain, Ω, is the set of all lines of code in all of the order-affecting algorithms.[6] However, not all order-affecting algorithms are ordered algorithms. A previously mentioned example is the calculation of molecular viscosity, μ, via an algebraic law in a Navier-Stokes code. There is not an associated OA for the μ calculation itself, but since μ is referenced in the term $\frac{\partial}{\partial x}\left(2\mu\frac{\partial u}{\partial x}\right)$, which does have an OA associated with its discretization, the μ calculation is an order-affecting algorithm and therefore must be verified. All ordered algorithms must have their observed OA measured via an OAT and compared with the target OA. However, it may be possible, under certain circumstances, to leave testing of some order-affecting algorithms which are not ordered algorithms to the unit-test suite, rather than including them in the OVTS. Continuing with the example of μ calculation, in the event that several different options exist (Sutherland's law, Keyes' law, and constant viscosity), it is not necessary to have separate OATs for each of these cases. At least one OAT must exist to test $\frac{\partial}{\partial x}\left(2\mu\frac{\partial u}{\partial x}\right)$ and one of the options for μ, in the most general way imaginable. Since Sutherland's law and Keyes' law are both functions of temperature, either could be used in order to satisfy this generality constraint, but constant viscosity could not be used because that would potentially overlook some coding mistakes. Take for instance a coding mistake which passed the μ function a temperature from node i–1 instead of node i. In the case of constant viscosity, this would not matter, and the mistake would go unnoticed. However, if Sutherland's law or Keyes' law were used, this would show up as an unordered error if the relative locations of nodes i–1 and i were uncorrelated or a first-order error if they were adjacent or otherwise correlated. Once an OVE for the Navier-Stokes equations with a general viscosity definition is completed, other viscosity options can be tested via unit tests, provided the software is designed such that no coverage gaps exist in the union of the OAT and the unit tests (e.g., minimal code duplication prevents lines from being missed). These unit tests are not considered to be part of the OVTS, since they are not functional (i.e., end-to-end) tests. However, these unit tests do cover some fraction of Ω and thus make a smaller Π_{OVE} permissible.

When OATs are used to fill the identified coverage gaps, then the issue of OVTS granularity resurfaces. The need for granularity may depend upon the identified gap: some portions of the OV domain may warrant fine granularity—others may not. In cases where fine granularity is desired, slight modifications to an existing OAT would create a new OAT having a high degree of overlap with the OAT from which it was derived. On the other hand, replacing an existing OAT with one that has a superset of OV domain coverage would be more appropriate if coarse granularity is desired.

An example OAT introduction which resulted in finer granularity involved two new convecting-vortex tests. The seven convecting-vortex tests in the initial OVTS used seven different temporal-integration schemes, but each used the same initial-condition (IC) enforcement option. Two alternative IC enforcement options were identified as being uncovered by the OVTS. In this particular case, two new tests were created from an existing one, so that each used the same temporal-integration scheme but a different IC enforcement option. This rendered the initial test redundant, since it shared the same IC enforcement option with six other tests and the same integration scheme with two other tests. However, since the test was already in the OVTS, and the granularity created by the overlap was desirable, this test was left in the OVTS. The effect of this and other

[6] Actually, it is the set of code lines defined by the order-affecting algorithms in all of the combinations of interest. However, for this study, we do not attempt to define which combinations are interesting or even possible. An example is verification of the BCs. Each BC can be adjacent to certain other BCs along junctions in the domain boundary. BC/BC coupling issues may arise at these junctions, but at present, we only consider whether a BC has been tested using a sufficiently general solution and sufficiently general mesh topologies, not whether all possible BC/BC interactions have been examined. Examination of the intricacies of BC/BC coupling is being postponed in Premo until the order behavior of present BCs is well understood and acceptable under uncoupled conditions.

granularity decisions on the progress and fitness measures is discussed in Sects. 4.1 and 4.2.

2.4 Finalizing the OVTS

Steps 2 and 3 (evaluating coverage of the OVTS and introducing new OATs to fill gaps) are repeated until the OVTS is complete. Since Ω is finite, the OVTS associated with $\Pi_{\mathrm{OVE}} = \Omega$ is also finite. However, under time and budgetary constraints, some portions of the OV domain are sometimes excluded from consideration, so the OVTS is declared to be complete relative to $\Pi_{\mathrm{OVE}} \subset \Omega$. As mentioned previously, this study did not attempt to completely cover portions of the OV domain mapped out by Sierra framework or TPL options. Nor did it attempt to fully cover portions of the OV domain mapped out by Premo capabilities that were experimental or deprecated. With regard to the mature and documented capabilities that were considered in OVTS construction, not all possible interactions and combinations were considered. Performance of iterative solvers was also not intended to be verified, so sections of code corresponding to Jacobian evaluations, nonlinear solvers, etc., were not considered as part of Ω; although the OATs did exercise some of the solvers and a variety of their options, their entire option sets are not covered by the OVTS.

The final OVTS for this study consisted of 52 tests: nine convecting-vortex tests for temporal-integration schemes and IC enforcement options, eight unsteady exact-solution tests for flux-limiter options, and 35 MMS tests for spatial verification of all of the options associated with the interior-equation sets and BCs. Not all of these OATs are complete, but each has all of the inputs defined (input file, mesh file, etc.) necessary to define its coverage in the OV domain. A ''complete'' OAT, according to [1], also has everything needed to actually run and post-process the tests. For example, the eight unsteady exact-solution tests for limiter options are incomplete because the appropriate exact solution has not been chosen or implemented; however, they do have all of the inputs defined to demonstrate that they cover the full set of limiter options.

3 Measuring progress relative to the OVTS

3.1 Measuring progress relative to the particular OV domain of the OVE, Π_{OVE}

Once the OVTS has been constructed, it is used to demonstrate the order behavior of the code. During this demonstration phase, progress can be measured by one of several measures suggested by [1]. The first of these is the status measure, S, and it is given as

$$S = \frac{1}{Np_{\max}} \sum_{n=1}^{N} p(l_n)$$

where N is the number of OATs in the OVTS, $p_{\max}$ is the maximum number of points assignable to a given OAT, l_n is the status level of the nth OAT, and $p(l)$ is a function which specifies how many points are assigned for each status level in this study.

Table 2 shows the different status levels and the points (out of a maximum of 20) awarded to the OATs at each level. For more details, see [1]. In later sections, when the function $p(l)$ is described as being unequally weighted, the point assignments in Table 2 are implied. Sometimes an equally weighted $p(l)$ is calculated, implying equal incremental progress from each level to the next, and sometimes a pass/fail $p(l)$ is calculated, implying full credit at level 5 and no partial credit for the lower levels.

The status measure is an excellent way to measure the effort level in the OVTS demonstration process and estimate the time required for OVTS completion, since it gives equal emphasis to each test. However, it is also useful to ask the following question: ''What fraction of the OV domain is covered by the reproducible OATs, and what fraction still remains?'' In order to answer this question, a decomposition for the OV domain must be chosen (input keywords, functions, lines, etc.), and the union of all particular domains associated with reproducible OATs, $\bigcup_{m=1}^{M} \Pi_m$, where M is the number of OATs at level 5, must be found. In this study, two such measures are calculated versus time for Premo:

Table 2 OVTS Demonstration process with OAT status levels

Level	OAT status	Required activity	Failure mode	Points
0	Incomplete	Determine all input	Incomplete input	0/20
1	Ready	Obtain solutions	Missing solutions	3/20
2	Realized	Demonstrate asymptotic OA	Non-asymptotic OA	5/20
3	Asymptotic	Verify observed OA is target OA	Unverified OA	10/20
4	Verified	Document, set up regression test	Non-reproducible	12/20
5	Reproducible	Proceed to next OAT		20/20

- $P_{\Pi,3}$, the ratio of the number of functions covered by reproducible OATs to the number of functions covered by the full OVTS.
- $P_{\Pi,4}$, the ratio of the number of input keywords covered by reproducible OATs to the number of input keywords covered by the full OVTS.

Since only reproducible OATs (i.e., those at level 5) are counted, a pass/fail definition for $p(l)$ is implied. For these measures to be meaningful, the OVTS must be complete relative to the full OV domain, Ω. These measures are discussed in more detail in Sect. 4.1. There are other measures given in [1] which will be evaluated for Premo in future studies.

3.2 Measuring progress relative to the particular OV domain of an application, $\Pi_{\text{app.}}$

The previously mentioned measures (S, $P_{\Pi,3}$, and $P_{\Pi,4}$) measure progress in OV relative to Π_{OVE}. It is also useful to know the level of verification progress relative to the particular OV domain of an application, $\Pi_{\text{app.}}$. The fitness measure introduced in [1] is given as

$$F = \frac{1}{p_{\max}} \sum_{n=1}^{N} w_n p(l_n)$$

where w_n is a weight assigned to the nth OAT indicating its relevance to the application. These weights can be determined manually or by some automated indicator of OAT relevance and overlap. If a decomposition for $\Pi_{\text{app.}}$ is chosen, then the weights can be determined by the extent of overlap of each OAT's particular domain, Π_n, with $\Pi_{\text{app.}}$. This is essentially a projection operation from Π_{OVE} to $\Pi_{\text{app.}}$ in the chosen decomposition. If functions are used for the decomposition, the fitness measure $F_{\Pi,3}$ (similar to the progress measure $P_{\Pi,3}$) is obtained, and if input keywords are used, the fitness measure $F_{\Pi,4}$ (similar to the progress measure $P_{\Pi,4}$) is obtained. Results for both of these measures are given in Sect. 4.2.

4 Results

4.1 Status-measure results

Figure 1 shows the progress of Premo verification over a 2.5 year period using status measures based on three different weightings: equally-weighted, unequally-weighted, and pass/fail. The equally-weighted status measure shows continuous verification progress over this time period. The jump in the summer of 2005 corresponds to the introduction of the first seven of the nine convecting-vortex exact-solution tests. These tests were conceived, implemented, and passed in a very short period of time. This step function is superimposed on the nearly linear progress of the MMS verification effort. The unequally-weighted status measure shows the same trends, only it is consistently lower because less weight is given to the lower OAT status levels (where most of the current tests reside). The pass/fail status measure does not show incremental progress over time—rather, it shows sudden jumps as OATs pass. The pass/fail weighting, however, provides the ultimate measure of success, because all OATs eventually need to pass.

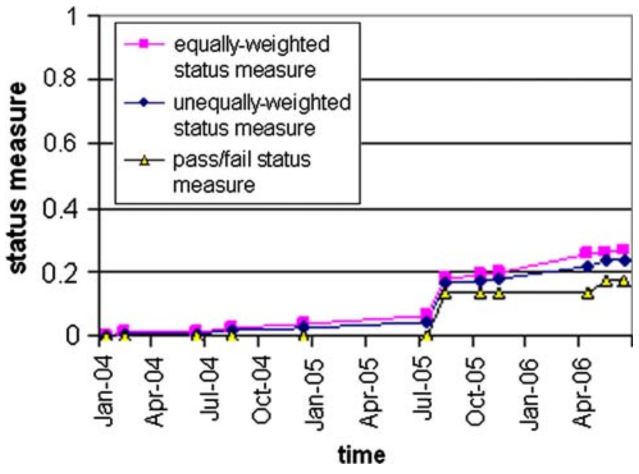

Fig. 1 Status measures versus time

The MMS tests, because they are more general than the convecting-vortex tests, cover larger fractions of the OV domain (or at least cover fractions more generally), but this is not reflected in the status measures shown in Fig. 1, since all OATs are weighted equally, without regard to how much of Π_{OVE} each OAT covers and whether this coverage is unique. If the tests are weighted based on their unique coverage attributes, then the progress measures obtained show much more significant progress and more accurately reflect the contributions of the MMS tests, which cover larger portions of the OV domain than the convecting-vortex tests.

Two coverage-weighted progress measures are shown in Fig. 2. Unfortunately, two problems are evident with these coverage-weighted measures: first, the weightings based on keyword and function coverage result in significantly different measures, even though they are intended to measure the same thing, and second, both are higher than the developers' perception of verification progress. The reason for these discrepancies is that keyword-based and function-based decompositions are, in practice, really incomplete representations of the OV domain. There are tests which may use option keywords or call functions but not test them in the most general fashion. Since the automated tools for determining keyword and function coverage do not account

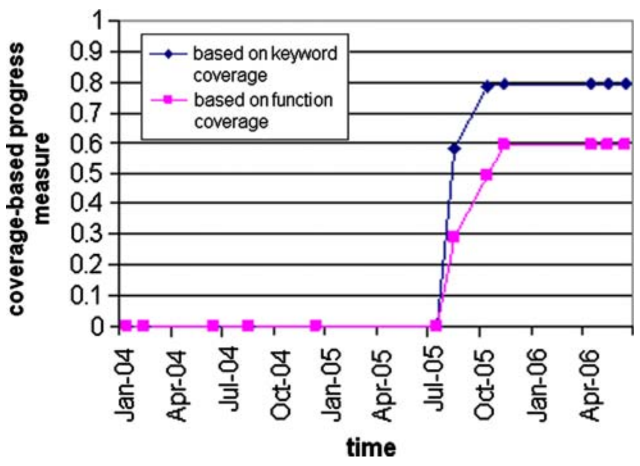

Fig. 2 Coverage-based progress measures versus time

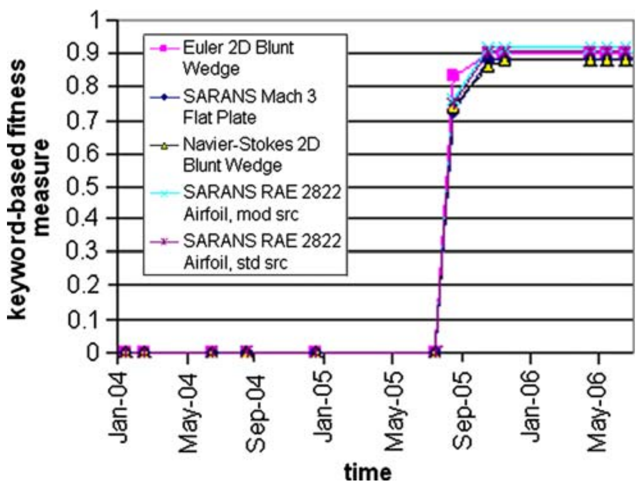

Fig. 3 Input-keyword-based fitness measure versus time

for this, the progress measures increase proportional to all of the unique keywords and functions of a test, rather than only increasing for the keywords and functions corresponding to features tested with all necessary generality. It does appear for this case that the function-based progress measure is less inflated than the keyword-based progress measure, but that may not be true in general. The explanation for the specific case of Premo is that there are many input file lines which are used by almost any test (setting the CFL number, number of time steps, etc.), so the first passing OAT causes a large increment in keyword-based progress measure.

4.2 Fitness-measure results

The fitness measure is a description of the code's verification status relative to a specific application. Since each application has an associated particular domain, $\Pi_{\text{app.}}$, then any progress-measure definition can be converted into a fitness measure by projecting it from Π_{OVE} to $\Pi_{\text{app.}}$. A more complete discussion of what is implied by the fitness measure is contained in [1]. Figure 3 shows the fitness measure for five different applications with the full OV domain and the subset for each application decomposed by input keywords. Figure 4 shows the fitness measure for three of these applications with a function-based partition for the full OV domain, Ω, and the subset for each application, $\Pi_{\text{app.}}$. As was the case in Fig. 2, both measures seem a little inflated, with the keyword-based measure being more inflated than the function-based measure. However, in the case of fitness measures, coverage-weighted measures are not guaranteed to be inflated, as they are for the progress measures. Whereas the approximate decomposition of Ω tends to inflate the measure, the approximate decomposition of $\Pi_{\text{app.}}$ tends to deflate the measure. Just as an OAT may not test a feature with all possible generality, an application may not use a feature with all possible generality (and therefore, not require it to be tested with all possible generality).

4.3 Discussion of granularity effects on progress measures

As was mentioned in Sects. 2.1 and 2.3, the granularity of the OVTS has a significant impact on the progress and fitness measures. A few trends in the plots bear witness to this effect. Figures 2, 3, and 4 show measures of zero up until the first test in this OVTS passed in August 2005. However, there were OATs created and passed prior to August 2005 that are not included in this OVTS (namely, the 2D results in [6, 7] and some inviscid results from [3–5]). The passing of these OATs is not reflected in any of the plots because these OATs are omitted from the OVTS. For this case, the coarser OVTS shows lower measures than

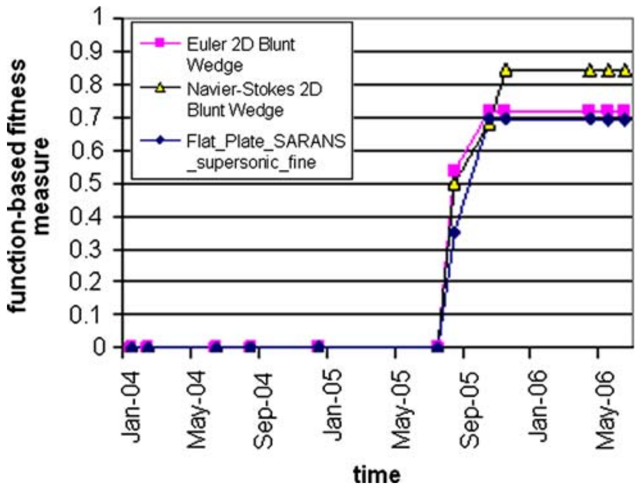

Fig. 4 Function-based fitness measure versus time

would an OVTS containing every OAT which has been passed.

Another granularity-induced anomaly is visible in Fig. 4, where Premo appears to have a higher readiness level for the Navier-Stokes (viscous) 2D Blunt Wedge application than for the Euler (inviscid) 2D Blunt Wedge application. This seems paradoxical, since all of the terms in the interior equations for the inviscid application are needed for the viscous application, but not vice versa. In this case, one of the OATs in the OVTS is an MMS test of the Navier-Stokes equations with all Dirichlet BCs. There is not an MMS test of the Euler equations with all Dirichlet BCs in this OVTS—instead, coverage of the Euler equations is gained through coverage of the Navier-Stokes equations and through Euler OATs which test more complicated BCs. However, the Euler tests with more complicated BCs had not yet passed at the time the fitness of Premo 1.3β was measured, and the Navier-Stokes OAT, though testing the same equation terms, does not call all of the same functions as would an Euler test. Thus, some Euler functions show up as being needed by the Euler 2D Blunt Wedge application but not covered by the passed portion of the OVTS. This anomaly could be prevented in future versions of the OVTS by including an Euler test with all Dirichlet BCs, or by re-organizing the function calls so that the Navier-Stokes test relies more heavily on the same functions as an Euler test would.[7]

Granularity and other properties of the OVTS can sometimes have a more profound impact on fitness measures than on progress measures. Some OATs within any given OVTS will overlap the OV subdomain associated with a given application. These can be referred to as the ‘‘relevant'' OATs for the application. However, the OV subdomain of the relevant OATs, $\Pi_{\text{rel.}} \subseteq \Pi_{\text{OVE}}$, will most likely be larger than the application-associated OV subdomain, $\Pi_{\text{app.}}$, because the boundaries of the OAT-associated subdomains will not coincide with the application-associated subdomains. In such a case, a coding mistake or algorithmic weakness could lie in a portion of the OV domain that is inside $\Pi_{\text{rel.}}$ but outside $\Pi_{\text{app.}}$. This coding mistake or algorithmic weakness would lower the fitness measure, even though it is not associated with a code capability needed for the application.[8]

This concept is illustrated in Fig. 5, where, for both the fine-grained and coarse-grained images, Π_{OVE} is represented by the outer rectangle; $\Pi_{\text{app.}}$ is represented by the inner circle, and coding mistakes or algorithmic weaknesses (which would reduce the observed OA) are represented by triangles. In the case of the fine-grained OVTS, the relevant OATs are represented by five smaller rectangles (the union of which is $\Pi_{\text{rel., fine}}$), and in the case of the coarse-grained OVTS, the relevant OATs are represented by three larger rectangles (the union of which is $\Pi_{\text{rel., coarse}}$). Two of the three relevant OATs for the coarse-grained OVTS would fail, resulting in a lower fitness measure than would be observed for the fine-grained OVTS, where all five OATs would pass. Thus different test suites result in different fitness measures for the same version of the code, same two order-reducing issues, and same application. Because of this potential, an OVTS which is finer-grained is more likely to produce an accurate fitness measure. This is, of course, only relevant in the transient situation where unresolved order-reducing issues are still present—the closer a code is to being fully verified, the less it matters whether OATs and applications are aligned in the OV domain.

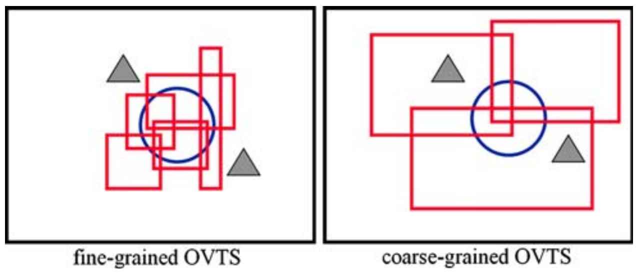

Fig. 5 Effect of granularity on fitness measure

In the preceding discussion, we consider various issues assuming that the criteria for granularity decisions were chosen with noble intentions. There remains, however, the possibility that one could refine the OVTS (or even worse, intentionally introduce redundancy) in portions of Π_{OVE} where a code's order behavior is known to be satisfactory and coarsen the OVTS in portions of Π_{OVE} where the code's order behavior is known to be unsatisfactory. This would falsely and deceptively inflate the measures. Although the coverage-based measures would filter for this somewhat, giving no credit for redundant OATs and less credit for fine-grained OATs, the coverage-based measures are typically already inflated because they can give credit for lines in Π_{OVE} which are not tested generally.

5 Conclusions

This practical illustration of [1] has shown that the theoretical ideas there translate well to measuring the order-verification progress of an actual code. Implementation of

[7] Additionally, this problem has been resolved in version 1.4β since many of the Euler-BC-associated OATs pass and also verify the implementation of the Euler equations on the interior of the domain; however, because this study focuses on the static code case, all of the plots reflect the performance of version 1.3β and its associated OVTS.

[8] This situation is recognizable only in hindsight, since a failing OAT will not identify where in the OAT-associated subdomain the coding mistake or algorithmic weakness lies.

theory will vary from code to code depending on the specifics of the code in question. For Premo, the main issue was how to implement a method for OVTS construction.

Many issues associated with OVTS construction and measuring progress in code OV have been addressed. A need has been identified for multiple OVTS coverage indicators to aid in OVTS construction. Also, the function-based and keyword-based indicators have been developed to supplement expert knowledge. Although these are not the only conceivable indicators of OVTS completeness, it can be stated in general that using multiple coverage indicators decreases the likelihood of missing a hole in OVTS coverage. The simple act of generating an OVTS and evaluating it for completeness may identify coding mistakes before any OAT is actually run. An example encountered in this work involved a logic error causing the proper function not to be called. When the input keyword associated with this feature was used, the feature showed up as covered by the keyword-based indicator, but not by the function-based indicator. The bug was corrected, and when the proper code path was taken, the function-based indicator reflected coverage of the feature. Once a complete OVTS has been created, the OATs can be prioritized based on their unique coverage attributes, as determined by these indicators. A thorough discussion of OAT prioritization is given in [1]. Although Premo's OV is not 100% complete, the formal process of constructing a complete OVTS identifies a clear path forward for completing its OV.

The non-uniqueness of the OVTS has a significant impact. In particular, the granularity of the OVTS can affect progress measures and, to an even greater extent, fitness measures. This granularity also affects the overall size and runtime of the OVTS, with a coarser-grained test suite generally being smaller and running faster than a finer-grained test suite. On the other hand, a finer-grained test suite has better diagnostic effectiveness, since the pass/fail status of individual tests can more accurately be mapped back to a list of features or portions of the software. In order to fit the changing needs of a code development project, the OVTS can be refactored periodically to increase or decrease its granularity. Granularity can be increased for portions of the OV domain that have evaded verification for extended periods of time, while granularity can be decreased for portions of the OV domain representing mature capabilities. Coarsening is especially important as the OVTS evolves from a tool for initial OV to a tool for regression-style OV. Also, the OVTS can also be refactored so that the boundaries of its OAT-associated OV subdomains are more aligned with application-associated OV subdomains. This would allow more accurate calculation of fitness measures for the applications of interest.

Even though the progress and fitness measures presented in [1] are OVTS-dependent, this study demonstrates that they are useful for measuring OV status and OV status relative to specific applications or classes of applications (i.e., fitness). The values presented are not intended to create a false sense of precision; rather, they are to provide data for discussion of relative OV progress between code developers and stake holders. Additionally, this study has created a database of OATs and associated OVEs which can be used for yet-to-be-determined measures. Lessons learned in OVTS construction will allow future OVTS versions to be superior to the current one, and lessons learned in OVTS completeness evaluation will lead to the development of better coverage indicators. Once an OVE or several OVEs have begun, calculating progress measures is relatively cheap, so using several different ones is feasible and may be a preferred way to adequately indicate how much progress in OV has been made and how much work remains. Some lessons learned affect the code design/development process. Generating an accurate mapping between the software, user options, and governing equations can make OVTS construction and evaluation much easier, and this mapping is easiest to generate as the software is being developed, not afterward.

In some cases, modular design of the software may allow some portions of the full OV domain, Ω, to be covered by unit tests, thus shrinking the particular domain remaining to be verified by an OVE, Π_{OVE}. A previously mentioned example is that of the viscosity calculation, in which case the most general option needs to be tested by the OVTS, and other options may be tested via unit tests and thus excluded from Π_{OVE}.

6 Remaining issues

Several issues discussed in [1] have not yet been put into practice for Premo. These issues include extending the OVTS to include interactions of different features (e.g., BC/BC coupling), measuring progress in OV for a non-static code, OAT prioritization, application-specific OVEs, and OVTS design criteria more suitable for regression testing. All of these issues are important, and the authors plan to investigate them in the near future using Premo. Additionally, even though the authors still do not plan to use line coverage to evaluate completeness of the OVTS, some of the coverage-based progress and fitness measures in [1] rely on line coverage. The authors have hypothesized that a line-coverage based progress measure, $P_{\Pi,5}$, would not be as falsely inflated as the function-coverage and input-keyword-coverage based progress measures, $P_{\Pi,3}$ and $P_{\Pi,4}$. Future work will examine this hypothesis. Other progress and fitness measures are presented in [1] which have not yet been calculated for Premo, so future endeavors will examine those measures as well.

Acknowledgments The authors would like to thank Jeffrey Fisher and Alfred Lorber for their assistance with the function-based coverage indicator. The authors would also like to thank Tolulope Okusanya for his assistance with the input-keyword-based coverage indicator and Thomas Smith for his assistance in identifying the mappings between governing equations, input options, and portions of the software.

References

1. Knupp PM, Ober CC, Bond RB (2007) Measuring progress order-verification within software development projects. Engineering with Computers (appears in this issue)
2. Smith TM, Ober CC, Lorber AA (2002) SIERRA/Premo-A new general purpose compressible flow simulation code. In: 32nd AIAA fluid dynamics conference and exhibit, St. Louis, Missouri, June 24–26 2002. AIAA-2002-3292
3. Bond RB, Knupp PM, Ober CC (2004) A manufactured solution for verifying CFD boundary conditions. In: 42nd AIAA aerospace sciences meeting and exhibit, Portland, Oregon, June 28–July 1 2004. AIAA-2004-2629
4. Bond RB, Knupp PM, Ober CC (2005) A manufactured solution for verifying CFD boundary conditions, Part II. In: 43rd AIAA aerospace sciences meeting and exhibit, Reno, Nevada, Jan. 10–13 2005. AIAA-2005-0088
5. Bond RB, Ober CC, Knupp PM (2006) A manufactured solution for verifying CFD boundary conditions, Part III. In: 36th AIAA fluid dynamics conference and exhibit, San Francisco, CA, June 5–8 2006. AIAA-2006-3722
6. Roy CJ, Ober CC, Smith TM (2002) Verification of a compressible CFD code using the method of manufactured solutions. In: 32nd AIAA fluid dynamics conference and exhibit, St. Louis, Missouri, June 24–26 2002. AIAA-2002-3110
7. Roy CJ, Nelson CC, Smith TM, Ober CC (2004) Verification of Euler/Navier-Stokes codes using the method of manufactured solutions. Int J Numer Methods Fluids 44:597–620
8. Edwards HC, Stewart JR (2001) SIERRA : a software environment for developing complex multiphysics applications. In: Bathe KJ (ed) Proceedings of first MIT conference on computational fluid and solid mechanics. Elsevier, Amsterdam, June 2001